VŠB - Technická univerzita Ostrava

Fakulta elektrotechniky a informatiky

Katedra Informatiky

Akcelerace algoritmů zpracování obrazu v prostředí NVIDIA CUDA

Acceleration of Image Processing Algorithms with NVIDIA CUDA

2011/2012 Michal Čerbák

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Michal Čerbák**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Akcelerace algoritmů zpracování obrazu v prostředí NVIDIA CUDA
Acceleration of Image Processing Algorithms with NVIDIA CUDA

Zásady pro vypracování:

Student by se měl seznámit se základními algoritmy používanými při zpracování obrazu (úprava jasu, hledání hran v obraze, rozmazání, inverze, zvětšování a využití např. konvoluční masky v těchto algoritmech) a s prostředím NVIDIA CUDA.
Práce by měla obsahovat základní popis rozhraní NVIDIA CUDA, jeho vývoj a základní vlastnosti (architektura multiprocesorů, paměti). Další část práce by se měla věnovat konkrétní implementaci v prostředí CUDA, očekávám vysvětlení použitých postupů a srovnání výsledků CPU a GPU kódu. V závěru by student měl zohlednit přínosy rozhraní NVIDIA CUDA, jeho pozitiva i omezení.

Seznam doporučené odborné literatury:

CUDA, Supercomputing for the Masses (http://www.drdobbs.com/high-performance-computing/207200659)
NVIDIA CUDA Compute Unified Device Architecture (http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf)

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

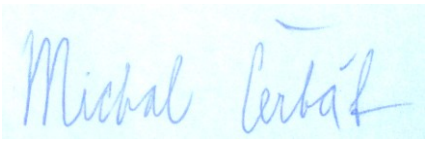
Vedoucí bakalářské práce: **Ing. Milan Šurkala**

Datum zadání: 18.11.2011
Datum odevzdání: 04.05.2012

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.



V Ostrave 25. 4. 2012 .....

.....

Michal Čerbák

Týmto by som sa chcel poďakovať svojmu vedúcemu Ing. Milanovi Šurkalovi za to, že mi umožnil sa zoznámiť s popisovanými technológiami a metódami a aj za jeho rady a pripomienky, ktorými mi pri práci na tomto texte pomáhal.

Abstrakt

Táto práca popisuje technológiu CUDA a teóriu metód spracovania obrazu a venuje sa ich spracovaniu sekvenčne a aj paralelne v prostredí CUDA. Medzi popisované metódy patria: úprava jasu, detekcia hrán v obraze, rozmazávanie, inverzia, zväčšovanie a preklápanie. Práca sa zaoberá aj porovnaním spracovaných algoritmov a popisuje ich rozdiely a relatívne zrýchlenie sekvenčného a paralelného algoritmu.

Kľúčové slová

CUDA, paralelizácia, spracovanie obrazu, úprava jasu, detekcia hrán, rozmazávanie, inverzia, zväčšovanie, preklápanie.

Abstract

This paper describes CUDA technology and theory of image processing methods and processes their parallel and sequential processing in CUDA. Described methods include: brightness, edge detection in an image, blur, invert, zoom and flipping. Paper also focuses on comparing processed algorithms and describes their differences and relative acceleration of sequential and parallel algorithm.

Key words

CUDA, parallelization, image processing, brightness, edge detection, blur, invert, zoom, flipping.

Zoznam použitých skratiek a symbolov

Skratky

CUDA	-	**C**ompute **U**nified **D**evice **A**rchitecture
CPU	-	**C**entral **P**rocessing **U**nit
DRAM	-	**D**ynamic **R**andom **A**ccess **M**emory
GPGPU	-	**G**eneral **P**urpose computation on **G**raphics **P**rocessing **U**nit
GPU	-	**G**raphics **P**rocessing **U**nit
HD	-	**H**igh **D**efinition
MHz	-	**M**ega **H**ertz
OS	-	**O**perating **S**ystem
SDK	-	**S**oftware **D**evelopment **K**it

Symboly

p(x, y)	-	funkcia pre prístup k bodu na súradniciach x a y

Obsah

1 Úvod ... 1

2 História CUDY ... 2

3 Architektúra CUDA ... 3

3.1 Základný popis ... 3

3.2 Práca v CUDE ... 3

3.2.1 Program ... 3

3.2.2 Vlákna ... 3

3.3 Hardwarová Architektúra ... 5

3.3.1 Compute Capability ... 5

3.3.2 Popis a princíp pracovania multiprocesoru ... 5

3.3.3 Rozdelenie Pamätí ... 6

4 Praktická časť ... 10

4.1 Invertovanie farby ... 11

4.1.1 Teória ... 11

4.1.2 Sekvenčná implementácia ... 11

4.1.3 Implementácia v CUDE ... 11

4.1.4 Výsledky ... 12

4.2 Zmena jasu ... 12

4.2.1 Teória ... 12

4.2.2 Implementácia ... 13

4.2.3 Výsledky ... 13

4.3 Algoritmus preklápania ... 13

4.3.1 Teória ... 13

4.3.2 Sekvenčná implementácia ... 14

4.3.3 Implementácia v CUDE ... 14

4.3.4 Výsledky ... 14

4.4 Detekcia hrán v obraze ... 15

4.4.1 Teória ... 15

4.4.2 Sekvenčná implementácia ... 16

4.4.3 Implementácia v CUDE ... 16

4.4.4 Výsledky 16
4.5 Bilineárna interpolácia 17
4.5.1 Teória 17
4.5.2 Sekvenčná implementácia 18
4.5.3 Implementácia v CUDE 19
4.5.4 Výsledky 21
4.6 Gaussovho rozostrovanie 21
4.6.1 Teória 21
4.6.2 Sekvenčná Implementácia 22
4.6.3 Implementácia v CUDE 23
4.6.4 Výsledky 26
5 Záver 28
6 Použitá literatúra 29
7 Zoznam príloh 30

1 Úvod

V dnešnej dobe je na trhu k dispozícii veľa programov, ktoré disponujú množstvom nástrojov na úpravu obrazu. Od zmeny rozlíšenia cez automatickú korektúru farieb, až po veľmi zložité filtre, ktoré dokážu dať aj nekvalitným obrázkom nádych profesionality. Tieto úpravy sú mnohokrát veľmi výpočtovo náročné a dokážu skutočne vyťažiť aj moderné 4 a viac jadrové procesory. V mojej práci som sa snažil spracovať bežné metódy využívané na úpravu obrazu do prostredia NVIDIA CUDA a využiť tak výkon grafického akcelerátoru, ktorý je pri práci na počítači mnohokrát úplne nevyužitý. Nové architektúry ako Fermi a v dobe písania práce predstavený Kepler poskytujú masívny paralelný výkon, ktorým by sa dali operácie vykonávané procesorom mnohonásobne zrýchliť.

Druhá kapitola je venovaná histórii technológie CUDA. Preberiem verzie vydaných SDK, zásadné zmeny, ktoré sa v architektúre udiali a aj najdôležitejšie nástroje, ktoré nám pomáhajú pri práci s CUDOU.

Tretia kapitola sa zaoberá vlastnosťami CUDY vo všeobecnosti. Rozoberiem v nej štruktúru programu pracujúceho s CUDOU, hierarchiu vlákien a blokov, princíp paralelizmu. Ďalej popíšem princíp fungovania paralelizmu v CUDE, popíšem načo slúži warp, z čoho sa skladá multiprocesor a predstavím a popíšem aj pamäte, ktorými disponuje CUDA zariadenie.

V štvrtej kapitole sú popísané konkrétne metódy spracovania obrazu. Od tých jednoduchších, ako je invertovanie farby a zmena jasu, až po tie zložité, ako je bilineárna interpolácia, alebo Gaussove rozostrovanie. Každá z metód je popísaná teoreticky. Ďalej nasleduje návrh sekvenčného a paralelného riešenia a popis postupov, ktoré boli pri návrhu využité. Pre obe riešenia sú namerané časy vykonávania, aby mohlo byť určené relatívne zrýchlenie, ktoré je na konci popísané.

V piatej kapitole, závere, je zhodnotený prínos mojej práce a aj aj technológie CUDA vo všeobecnosti.

2 História CUDY

Technológia CUDA[1] bola vyvinutá spoločnosťou NVIDIA. Tento projekt bol predstavený spoločne s 8. radou grafických kariet GeForce v novembri 2006. Prvá dostupná beta verzia CUDA SDK bola vypustená vo februári 2007. Verzia 1.0 bola vypustená v júni 2007, spoločne s novou radou grafických procesorov TESLA. TESLA grafické procesory sú založené na jadre G80. Je to prvý grafický procesor z dielní NVIDIA, zameraný iba na GPGPU[1].

V decembri 2007 vychádza CUDA verzie 1.1 prinášajúca podporu CUDY základnými ovládačmi NVIDIA. Je to významný pokrok pre vývojárov CUDA aplikácií, vďaka čomu je pre spustenie CUDA aplikácie potrebný iba grafický procesor a základné ovládače od verzie 169.xx. Táto verzia prináša aj podporu prekrývania pamäťových prenosov výpočtami, podporu 64 bitových verzií Windows a podporu práce s viacerými grafickými akcelerátormi.

V roku 2008 je predstavená CUDA verzie 2.0 spoločne s novou radou GT200. Táto verzia prináša podporu double precision[2] hardwarovo podporovanú radou GT200. V novej verzii sa objavila podpora Windows Vista, Mac OS X, ďalej podpora 3D textúr a nový nástroj NVIDIA Visual Profiler. Tento nástroj umožňuje vývojárom väčšiu spätnú väzbu pre optimalizáciu CUDA C/C++ programov.

V roku 2010 je vydaná CUDA SDK verzie 3.0 spoločne s architektúrou Fermi. Prináša natívnu podporu double precision výpočtov, podporu rekurzie a ukazovateľov na funkcie. Zlepšené boli aj debugovacie a profilovacie nástroje.

V máji 2011 vychádza nová verzia CUDA SDK 4.0 a prináša predovšetkým unifikáciu pamäťových priestorov a vylepšenia podpory viacerých grafických akcelerátorov.

[1] Výpočty na grafickom procesore pre všeobecné použitie, teda nemusia slúžiť iba na vykresľovanie obrazu.
[2] 64 bitová premennú s desatinnou čiarkou.

3 Architektúra CUDA

3.1 Základný popis

Technológia CUDA využíva grafické akcelerátory z dielní NVIDIA. Ich architektúra poskytuje výborné možnosti paralelizácie, vzhľadom na obrovský paralelný výkon pri nízkej cene oproti cene porovnateľne výkonných procesorov.

CUDA dovoľuje vývojárom dosiahnuť obrovské zrýchlenia v algoritmoch, ktoré majú za úlohu spracovanie veľkého množstva dát, ako sú algoritmy spracovania obrazu - kvôli veľkému množstvu obrazových bodov. CUDA prináša zrýchlenia v širokom spektre aplikácií, ako sú napríklad: rozpoznávanie obrazu, prehrávanie HD videa v reálnom čase, jeho dekódovanie, komprimačné algoritmy, ale aj veľmi sofistikované aplikácie vo vedeckej sfére.

Prináša možnosť implementácie paralelizmov prostredníctvom známych jazykov, ako je C/C++, Fortran, Python alebo Microsoft .NET Framework. CUDA zakladá na paralelizme vlákien usporiadaných do bloku a možnosti paralelizmu celých blokov. To vedie programátorov na rozdelenie celého problému na hrubé dielčie časti, ktoré môžu byť riešené nezávisle a paralelne blokom vlákien a na rozdelenie každého dielčieho problému na jednoduchšie časti, ktoré môžu byť riešené paralelne vláknami v bloku. Toto rozdelenie dovoľuje vláknam spolupracovať pri riešení každého dielčieho problému a pritom dovoľuje škálovateľnosť. Každý blok vlákien musí byť nezávislý na iných blokoch, aby bolo možné ich riešiť v rôznom poradí, na rôznych jadrách či už v kooperácii, alebo paralelne. Takže, skompilovaný CUDA program dokáže bežať na zariadení a pritom počet fyzických jadier nemusí byť známy. Táto škálovateľnosť umožňuje CUDE pokryť celú škálu grafických akcelerátorov, od najbežnejších GeForce, až po profesionálne karty Tesla a Quadro.

3.2 Práca v CUDE

3.2.1 Program

Program využívajúci architektúru CUDA sa odlišuje od obyčajného C/C++ programu oddelením časti kódu, vykonávaného na grafickej karte (ďalej len zariadení), ktorý sa nazýva kernel[2] a kódu, ktorý je vykonávaný na procesore (ďalej len host). Kernel je funkcia, ktorá je spustená na zariadení paralelne N-vláknami. V kóde je definovaný kľúčovým príkazom `__global__`.

3.2.2 Vlákna

Každé vlákno má svoj index v rámci bloku, v ktorom sa nachádza, prístupný v kernely prostredníctvom premennej *threadIdx*, táto premenná predstavuje trojzložkový vektor. Vlákna môžu byť identifikované pomocou jednorozmerného, dvojrozmerného, alebo trojrozmerného

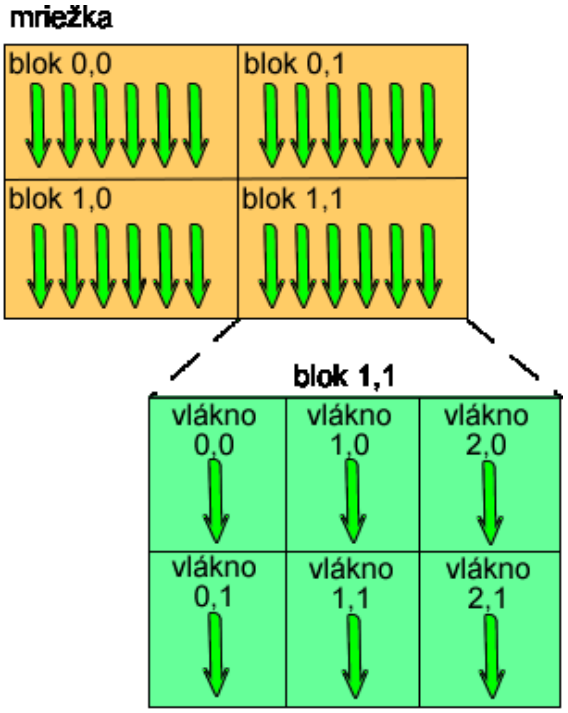
indexu. Tvoria teda jednorozmerné, dvojrozmerné, alebo trojrozmerné bloky. Táto hierarchia dovoľuje veľmi prirodzené pracovanie s viacrozmernými vektormi, maticami, alebo poľami.

Každé vlákno má svoje jedinečné ID v rámci bloku, v ktorom sa nachádza. Pri jednorozmernej konfigurácii je ID zhodné s indexom *threadIdx.x*, pri dvojrozmernej konfigurácii bloku o veľkosti *(Dx, Dy)* je ID vlákna rovné *(threadIdx.x + (threadIdx.y * Dx))*, pri trojrozmernej konfigurácii *(Dx, Dy, Dz)* je ID vlákna rovné *(threadIdx.x + (threadIdx.y * Dx) + (threadIdx.z * Dx * Dy))*.

Vlákna môžu spolu spolupracovať v rámci jedného bloku, tým že medzi sebou zdieľajú dáta v zdieľanej pamäti. Zdieľaná pamäť je umiestená blízko jadra procesoru, podobne ako L1 cache pamäť, preto dosahuje veľmi dobré latencie. Počet vlákien v jednom bloku je limitovaný práve nedostatkom pamäti Stream Multiprocesoru a jeden môže obsahovať maximálne 1024 vlákien.

Vlákna môžu synchronizovať svoje spúšťanie pomocou funkcie `__syncthreads()`. Tá má za úlohu pozastaviť beh vlákien na tomto bode, kým do tohto bodu nedorazia všetky vlákna: v tom momente činnosť vlákien pokračuje ďalej.

Bloky vlákien, podobne ako vlákna, môžu tvoriť jednorozmernú, dvojrozmernú, alebo trojrozmernú mriežku ako je na obrázku 1. Blok má svoj index uložený v premennej *blockIdx* obdobne ako vlákna, šírka jedného bloku je uložená v premennej *blockDim*. Rozmer mriežky je uložený v premennej *gridDim*, obe premenné predstavujú trojzložkový vektor. ID bloku sa počíta obdobne ako pri vláknach.

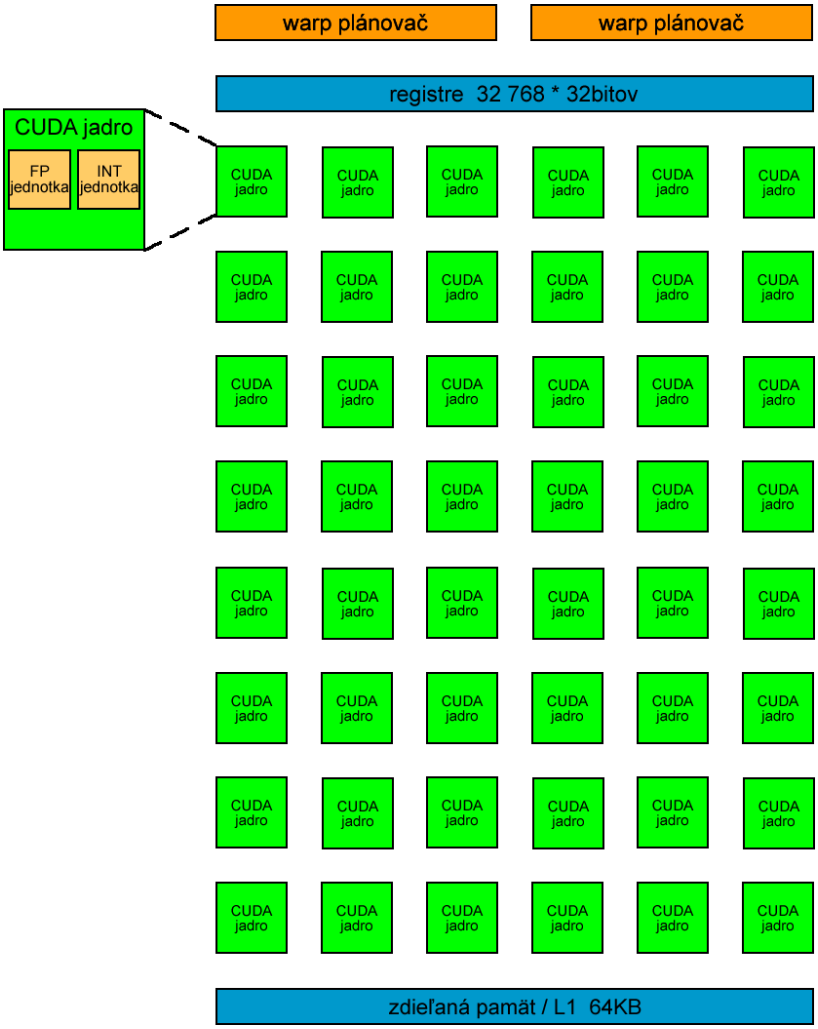


Obrázok 1. Ukážka rozloženia mriežky, blokov a vlákien

3.3 Hardwarová Architektúra

3.3.1 Compute Capability

Architektúra zariadení CUDA sa líši v závislosti na verzii Compute Capability[3][2] [3]. Pri práci na tomto texte som pracoval s kartou GeForce 560 Ti, ktorá má verziu 2.1 a preto sa budem venovať hlavne jej. Prvé číslo označuje architektúru, konkrétne číslo 2 znamená, že karta je založená na architektúre Fermi. Druhé číslo označuje menšie zmeny a vylepšenia architektúry. Na Obrázok 2 je znázornený jeden z 8 Stream Multiprocesorov tejto karty.



Obrázok 2. Grafické znázornenie Stream Multiprocesoru na grafickej karte GeForce 560 TI.

3.3.2 Popis a princíp pracovania multiprocesoru

Stream Multiprocesor vytvára, riadi a spúšťa vlákna, v skupinkách po 32 paralelných vlákien, ktoré sa nazývajú warpy. V chvíli, keď multiprocesor dostane jeden, alebo viac blokov vlákien

[3] Spoločnosť NVIDIA týmto pojmom označuje verziu architektúry danej grafickej karty.

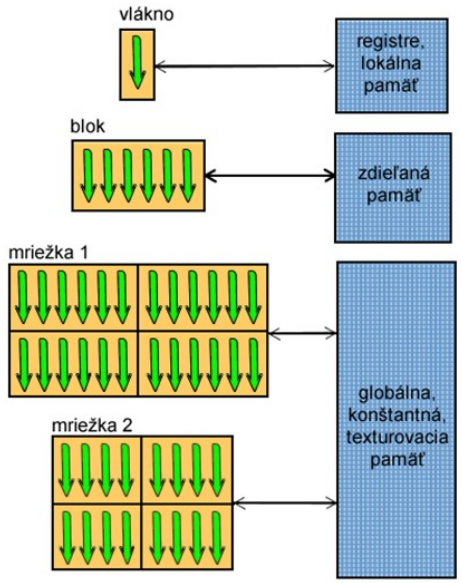
na vykonanie, prerozdelí ich do warpov, ktorých spúšťanie ďalej riadi warp plánovač. V jednom hodinovom cykle dokáže každý plánovač vydať dve inštrukcie pre warp, ktorý je pripravený ich vykonať. Jeden plánovač riadi warpy s nepárnym ID a druhý s párnym.

V zariadení s Compute Capability 2.1 sa nachádza 48 CUDA jadier, ktoré sú organizované do troch skupín po 16. Každá skupina 16 jadier je navrhnutá vykonať inštrukciu warpu (až na pár inštrukcií, ktoré zaberú viac) za dva hodinové cykly. Preto by priepustnosť mala dosahovať 3 vykonané inštrukcie warpu za dva hodinové cykly.

Najefektívnejšie vykonávanie warpu nastáva, ak všetkých 32 vlákien musí vykonať rovnakú inštrukciu. Podmienka v kóde, ktorá delí program na rôzne vetvy, môže spôsobiť odlišnosť inštrukcií jednotlivých vlákien, ktorá je nazývaná aj divergencia vlákien. V takom prípade warp postupne vykoná každú vetvu programu a zamaskuje vlákna, ktoré sa na danej vetve nepodieľajú. Ak sa teda program vetví na N–vetiev, warp sa spustí N–krát. Táto divergencia môže nastať len v rámci jedného warpu, iné warpy vykonávajú svoje inštrukcie nezávisle. Tento efekt sa dá obmedziť minimalizovaním použitia podmienok v kerneloch, prípadne skrátením podmienkových vetiev programu.

3.3.3 Rozdelenie Pamätí

Vlákna pri svojom behu môžu pristupovať do rôznych druhov pamätí, ako je to znázornené na Obrázok 3. Každé vlákno má k dispozícii vlastnú pamäť v podobe registrov a lokálnej pamäti, ktorá je dostupná len v rámci tohto vlákna. Blok vlákien má k dispozícii zdieľanú pamäť, prístup k tejto pamäti majú len vlákna v rámci jedného bloku. Všetky vlákna môžu pristupovať ku globálnej pamäti, táto pamäť je dostupná aj v rámci viacerých kernelov, podobne ako textúrovacia pamäť a pamäť pre konštanty, ktoré sú určené len na čítanie.



Obrázok 3. Hierarchia pamätí

Pamäť zariadenia je fyzicky rozdelená jej umiestením. Globálna pamäť, pamäť konštánt a textúrová pamäť sa nachádza v DRAM[4], zatiaľ čo registre, zdieľaná pamäť a cachovacie pamäte sa nachádzajú priamo na čipe.

3.3.3.1 Globálna pamäť

Globálna pamäť je najzákladnejšia pamäť, má veľkú kapacitu v podobe takmer celého DRAM priestoru a tvorí teda najväčšiu časť pamäti zariadenia. Má vysoké prístupové doby, v stovkách strojových cyklov. Compute Capability 2.x prináša cachovanie pamäťových transakcií v L1 a L2 cache, čo má za úlohu minimalizovať prístup do globálnej pamäte. Pre efektívnejší prístup do pamäti sa využíval združený prístup do pamäti a pre jeho dosiahnutie boli veľmi striktné pravidlá. Ale to už neplatí pre verziu Compute Capability 2.x, vlákna môžu k rôznym dátam pristupovať v rôznom poradí a aj pristupovať k tým istým dátam naraz, bez straty rýchlosti.

Globálna, rovnako ako konštantná a textúrovacia pamäť, musí byť alokovaná a naplnená ešte pred spustením kernelu. Host dokáže zapisovať a čítať iba z globálnej pamäti a z pamäti konštánt. CUDA ponúka množstvo metód pre alokovanie tejto pamäti, jej vyčistenie a kopírovanie.

Lineárna pamäť zariadenia je typicky alokovaná pomocou príkazu `cudaMalloc()` a dealokovaná pomocou `cudaFree()`, pre prenos medzi pamäťou zariadenia a pamäťou hostu sa typicky využíva funkcia `cudaMemcpy()`.

3.3.3.2 Pamäť konštánt

Konštantná pamäť sa nachádza v pamäti zariadenia. Používa sa pre konštanty a pre argumenty kernelu. Je relatívne pomalá, podobne ako globálna pamäť a je cachovaná v constant cache. Je optimalizovaná pre broadcast[5]. V kernely je prístupná iba na čítanie. Označuje sa kľúčovým slovom `__constant__`.

3.3.3.3 Pamäť textúr

Rovnako, ako konštantná, je táto pamäť súčasťou pamäte zariadenia. Tento typ pamäte je cachovaný do texture cache a tá je optimalizovaná pre 2D priestorové prístupy.

Ponúka rôzne adresácie prvkov, adresovacie módy a aj rôzne druhy filtrovania vyberaných hodnôt bez znižovania rýchlosti prístupu. Je veľmi vhodná pre využitie v aplikáciách pracujúcich s obrazom, zmenou jeho veľkosti a pod.

3.3.3.4 Registre

Registre sú najrýchlejšia pamäť, ktorú zariadenie poskytuje. Nachádzajú sa priamo v multiprocesore a a dokážu uložiť 32768 premenných s veľkosťou 4 bajty. Slúžia na ukladanie lokálnych premenných vytvorených v kernely. Vlákna teda môžu pristupovať iba k dátam, ktoré

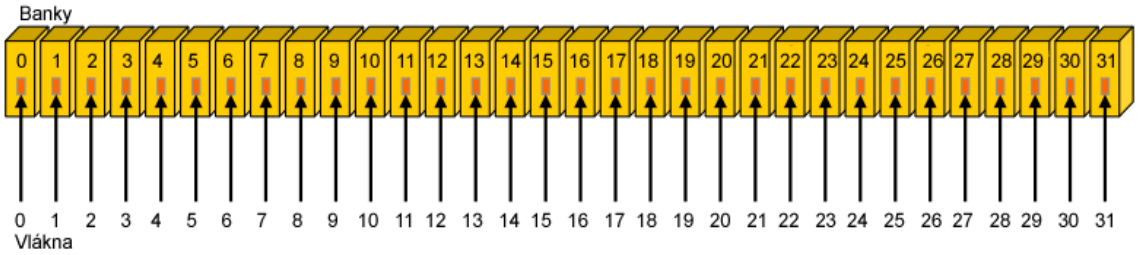
[4] Pamäť zariadenia s náhodným prístupom.
[5] Typ prístupu k pamäťovej bunke viacerými vláknami naraz.

sami vytvorili. Počet využívaných registrov v kernely dokáže veľmi ovplyvniť výkon zariadenia, pretože pri prekročení kapacity registrov sa ďalšie registre ukladajú do pomalej pamäte zariadenia, čo môže spôsobiť obrovské zvýšenie prístupových dôb.

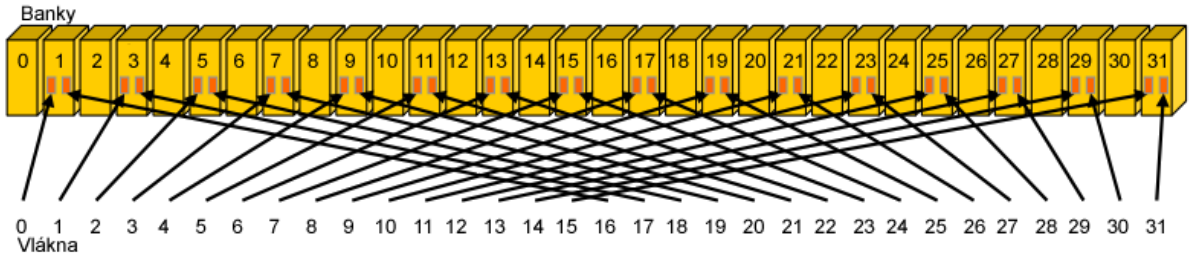
3.3.3.5 Zdieľaná pamäť

Zdieľaná pamäť je prístupná iba v rámci jedného bloku, ktorý ju dynamicky alokoval. Dosahuje vynikajúcu priepustnosť a latenciu v porovnaní s globálnou pamäťou, pretože sa nachádza na čipe spolu s L1 cache a delia sa o 64KB spoločnej pamäte v pomere 48KB/16KB, alebo naopak 16KB/48KB (pri nastavení tohto pomeru je potrebné určiť, koľko zdieľanej pamäte sa bude využívať, prípadne ako a ako často sa bude pristupovať ku globálnej pamäti). Kvôli dosiahnutiu veľkej šírky prenosu, je zdieľaná pamäť rozdelená do 32 modulov zvaných banky, do ktorých môžu vlákna pristupovať naraz. Dáta sú do bánk ukladané po 32 bitoch, ktoré sú nazývané slová (words). Pri uložení slova do poslednej banky sa ďalšie slovo začína ukladať znovu od prvej banky: je to tzv. prekladané ukladanie.

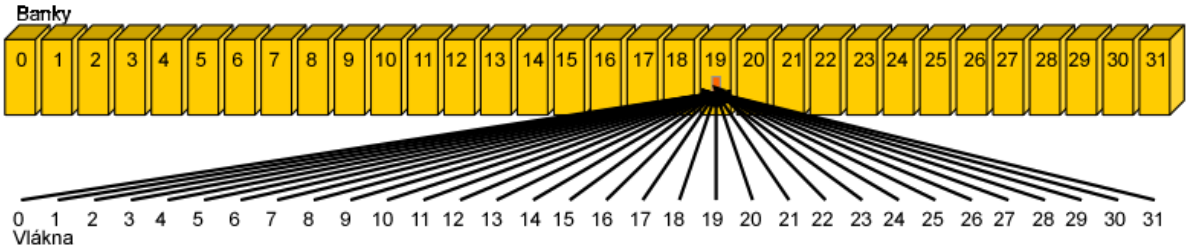
V prípade, že dve pamäťové žiadosti spadajú do jednej banky a zároveň smerujú na rozličné slová, vzniká tzv. bankový konflikt. Hardware teda rozdelí pamäťovú požiadavku na N-požiadaviek bez bankových konfliktov, ktoré budú vykonané postupne, tento konflikt je N-cestný. V prípade, že dve pamäťové žiadosti spadajú do jednej banky a odkazujú na rovnaké slovo, nevzniká žiadny bankový konflikt.



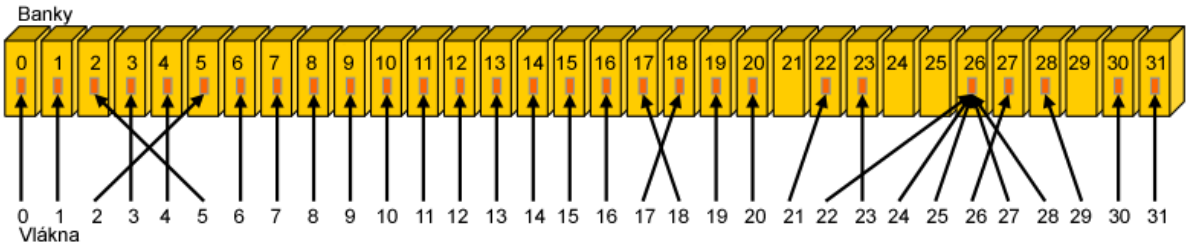
Obrázok 4. Lineárne adresovanie s posunom jedného slova. Sekvenčný prístup k bankám, bez konfliktov



Obrázok 5. Lineárne adresovanie s posunom dvoch slov, vlákna pristupujú k rôznym slovám v rovnakých bankách, vzniká 2- cestný bankový konflikt



Obrázok 6. Broadcastový prístup k jednej banke a rovnakému slovu, bez konfliktov



Obrázok 7. Náhodný prístup k bankám, vlákna 22, 24, 25, 27 a 28 pristupujú k rovnakej banke a rovnakému slovu, bez konfliktu

4 Praktická časť

Táto časť práce sa venuje určeným metódam spracovania obrazu. Každý z nich je teoreticky popísaný a ďalej je navrhnuté jeho spracovanie sekvenčne a paralelne v prostredí CUDA. Pri zložitejších riešeniach je popísaný aj vývoj riešenia. Pri každom z riešení je nameraný čas doby vykonávania na grafickej karte NVIDIA GeForce 560Ti s taktom grafického čipu 1645MHz a na procesore AMD Phenom II X4 s taktom 3400 MHz a vypočítané relatívne zrýchlenie. Aj tieto hodnoty sú pri jednotlivých metódach popísané. Ako vývojový nástroj bolo zvolené Microsoft Visual Studio 2010.

Meranie času

Pri meraní času som použil funkciu `rdtsc()`, ktorá zaznamenáva poradie aktuálneho cyklu procesoru. Na začiatku merania zaznamenávam hodnotu *start* a na konci hodnotu *end, F_CPU* je takt procesoru v Hertzoch. Pre výpočet výsledného času som použil vzťah *(end – start) / F_CPU*. Výsledný čas je v sekundách, pretože delíme celkový počet cyklov, počtom cyklov za jednu sekundu.

Pri meraní výsledných časov sa musí počítať s odchýlkami, ktoré spôsobuje predovšetkým nekonštantné vyťaženie systému- tieto odchýlky som minimalizoval počtom iterácií, ktoré sú následne spriemerované. Pre časovú náročnosť bol počet iterácií na procesore nižší– 1000 a na grafickej karte 10 000 iterácií. Pre väčšiu presnosť som používal aj takzvanú zahrievaciu iteráciu, ktorá sa do času nepočíta, tá má za úlohu zvýšiť takty zariadenia, ktoré sú v dobe nečinnosti znížené, kvôli šetreniu energie. Na procesore nebolo potrebné používať zahrievaciu iteráciu, pretože procesor mal v dobe merania vypnutú funkciu šetrenia energie. Do časov sa počítajú iba samotné algoritmy spracovania dát, nepočítajú sa operácie spojené s presúvaním dát z hostu do pamäti zariadenia a ani presun výsledku naspäť na host, ako ani zobrazenie výsledku.

Meranie časov som robil nad obrázkom s rozlíšením 512 x 512, 1024 x 1024, 2048 x 2048 pixelov. Časy uvádzané v tabuľkách sú v milisekundách. Relatívne zrýchlenie sa dá predstaviť, ako počet iterácií, ktoré dokáže zariadenie vykonať v čase jednej iterácie na procesore.

OpenCV

OpenCV je súbor knižníc, ktoré ponúkajú funkcie pre prácu s obrazom. Pri praktickej implementácii používam knižnice OpenCV vo verzii 2.1, na načítanie dát zo súborov obrázkov a následné zobrazenie výsledného obrázku.

Pre načítanie obrázku používam funkciu `cvLoadImage()`, ktorej parametrom predávam názov zdrojového obrázku a mód, ktorým bude obrázok načítaný. Pre zjednodušenie používam mód, ktorý otvorí a prekonvertuje farebné zložky obrázku na stupne šedi a prekonvertovaný obrázok je teda čiernobiely. Funkcia naspäť vracia inštanciu triedy `IplImage`, ktorá obsahuje

hodnoty jednotlivých pixelov. Tie sú typu unsigned char[6]a nadobúdajú hodnoty 0 (čierna) až 255 (biela). Pretože prístup k poľu pixelov je cez inštanciu triedy `IplImage` zbytočne zložitý, preukladám ich v cykle do nového jednorozmerného poľa, s ktorým je práca omnoho jednoduchšia.

Po vykonaní daného algoritmu je výsledné pole pixelov predané do novej inštancie `IplImage`, ktorá sa vytvára pomocou funkcie `cvCreateImage()`, aby bolo možné spracované pixely opäť zložiť na obrázok.

4.1 Invertovanie farby

4.1.1 Teória

Invertovanie, známe aj ako negatív, patrí medzi najjednoduchšie metódy spracovania obrazu. Predstavuje zmenu farby pixelov na ich opačnú hodnotu. Hodnota farby pixelu udáva vzdialenosť od čiernej farby, pri invertovaní má táto hodnota predstavovať vzdialenosť od bielej farby. Ak má napr. pixel hodnotu 50, tak je vzdialený 50 odtieňov od čiernej farby, po invertovaní bude tento pixel vzdialený 50 odtieňov od bielej farby. Aby sme to dosiahli, je potrebné od hodnoty predstavujúcej bielu farbu odčítať hodnotu spracovávaného pixelu. Výpočet je nasledovný:

$$p(x,y) = 255 - p(x,y) \qquad (1)$$

4.1.2 Sekvenčná implementácia

Sekvenčná implementácia postupne pristupuje k jednotlivým pixelom v poli dát pomocou dvoch cyklov: jeden na posun v riadku, ktorý je vnorený v druhom cykle na posun v stĺpci. Po načítaní hodnoty spracovaného pixelu je vypočítaná jeho invertovaná hodnota a tá je spätne uložená do poľa určeného pre spracované pixely.

4.1.3 Implementácia v CUDE

Vo verzii CUDA je potrebné najprv alokovať dve polia rovnakej veľkosti pre vstupný a výstupný obrázok o veľkosti *(šírka obrázku * výška * počet bajtov v jednom pixely)* pomocou funkcie `cudaMalloc()`, po alokovaní je potrebné pomocou funkcie `cudaMemcpy()` prekopírovať vstupné dáta z pamäte hostu do alokovanej pamäte v zariadení. Pri spracovaní v CUDE je potrebné správne rozloženie vlákien v bloku a blokov v mriežke, keďže vstupné pole pixelov je jednorozmerné, je v tomto prípade vhodné aj jednorozmerné rozloženie vlákien a blokov. Optimálne rozloženie vlákien v bloku je kľúčová časť konfigurácie kernelu. Vlákna v bloku by mali byť násobkom čísla 32, čo je počet vlákien vo warpe. Za predpokladu, že nevznikne divergencia vlákien, budú v takomto prípade všetky warpy plne obsadené. V praxi využívam 256 vlákien na jeden blok. Počet spracovávaných blokov počítam ako počet všetkých pixelov vydelený počtom vlákien v jednom bloku.

[6] Bezznamienková premenná s veľkosťou 1 bajt.

Pretože jednotlivé pixely a výpočty na sebe nie sú závislé, je tento algoritmus veľmi ľahko paralelizovateľný. Využívanie zdieľanej pamäte v tomto prípade stráca zmysel, pretože hodnota farby pixelu je využívaná iba raz v rámci jedného výpočtu a jedného vlákna. Pri výpočte sú všetky vlákna aktívne, pretože sa program nevetví a vlákno s jedinečným indexom ID spracováva pixel v jednorozmernom poli na pozícii ID, vďaka čomu je maximálne využitá metóda združeného prístupu do globálnej pamäte, ktorá čiastočne skrýva latenciu tejto pamäte. Kód kernelu sa nachádza v prílohe I.

4.1.4 Výsledky

	512 x 512	1024 x 1024	2048 x 2048
CPU	0,79	3,16	12,76
GPU	0,0542	0,0966	0,2599
Rel. zrýchlenie	14,6	32,7	49,1

Tabuľka 1.

Z nameraných časov je vidno, že táto metóda je skutočne výpočtovo nenáročná a aj sekvenčný algoritmus je veľmi rýchly. Ďalej je vidieť, že v sekvenčnom algoritme trvá výpočet jedného pixelu približne rovnako aj pri rôznych rozlíšeniach, ale v CUDE sa čas výpočtu jedného pixelu s narastajúcim rozlíšením znižuje. Je to spôsobené tým, že v časoch nameraných na CUDE je zarátaný okrem výpočtov aj čas spúšťania kernelu a réžia vykonávania kernelu. Čas spúšťania kernelu som zistil nameraním času vykonania prázdneho kernelu. Opäť som použil 10000 iterácií, pre väčšiu presnosť merania. Výsledný čas spustenia jedného kernelu je približne 0,0402 ms, počet vlákien a blokov čas spustenia neovplyvňuje. Po odpočítaní času potrebného na spustenie kernelu od nameraných časov je vidno, že samotný výpočet zaberie v prvom prípade iba 0,014 ms, je teda menej časovo náročný, ako samotné spustenie kernelu. Z tohto pohľadu je spúšťanie kernelu relatívne časovo náročné.

4.2 Zmena jasu

4.2.1 Teória

Zmena jasu predstavuje metódu spracovania obrazu, ktorá mení jas v obraze. Existuje veľmi veľa spôsobov zmeny jasu, ja som pri práci uvažoval nad dvomi: absolútnou a percentuálnou. Absolútna zmena jasu funguje tak, že k hodnote každého pixelu je pričítaná zadaná hodnota a z tohto dôvodu môže nastať prípad, kedy bude výsledná hodnota prevyšovať maximálnu hodnotu pixelu a bude nutné použitie podmienok. Druhý spôsob je percentuálna zmena jasu, ktorá vypočíta rozdiel maximálnej hodnoty pixelu a jeho skutočnej hodnoty a z neho vypočíta percentuálnu časť. Sčítaním percentuálnej časti a skutočnej hodnoty pixelu vzniká hodnota pixelu po zmene jasu. Pričítavanie percentuálnej časti zvyšku do maxima zaručuje, že výsledok patrí do intervalu <0, 255>, čo je veľmi výhodné pre implementáciu v CUDE, pretože sa vyhýbame podmienkam a teda aj možnej divergencii vlákien. Preto bol pre spracovanie vybraný druhý spôsob. Tento vzťah predstavuje výpočet jedného pixelu:

$$p(x,y) = \frac{(255 - p(x,y)) * percento}{100} + p(x,y) \quad (2)$$

Pomocou percenta si vypočítame percentuálny podiel z invertovanej hodnoty transformovaného pixelu, ktorú sčítame s jeho hodnotou a dostávame výslednú hodnotu so započítanou zmenou jasu.

4.2.2 Implementácia

Princíp sekvenčného a paralelného algoritmu je (až na výpočet transformácie pixelu) rovnaký, ako v algoritme invertovania farby popísanom v podkapitole 4.1 a preto nemá zmysel ho znovu popisovať. Kód kernelu sa nachádza v prílohe II.

4.2.3 Výsledky

	512 x 512	1024 x 1024	2048 x 2048
CPU	3,58	14,74	59,27
GPU	0,0564	0,1033	0,2846
Rel. Zrýchlenie	63,5	142,7	208,3

Tabuľka 2.

V tabuľke výsledkov je vidno, že algoritmus je dobre paralelizovateľný a v porovnaní s algoritmom invertovania, popísanom v podkapitole 4.1, dosahuje vyššie zrýchlenia, pretože používa viac numerických výpočtov, na ktoré je CUDA stavaná. Vysoké zrýchlenie sa dostavuje už pri obrázkoch s nízkym rozlíšením a so zvýšením rozlíšenia sa zvyšuje aj zrýchlenie.

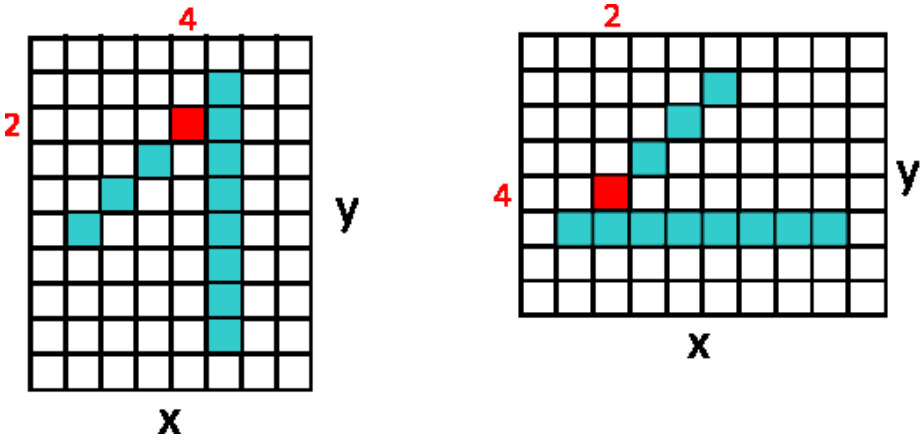
4.3 Algoritmus preklápania

4.3.1 Teória

Tento algoritmus patrí k kým najjednoduchším a pre ilustráciu metódy, ktorá sa nedá veľmi dobre paralelizovať je veľmi vhodný. Princíp transformácie môže byť vyjadrený vzťahom:

$$p(x,y) = p(y,x) \quad (3)$$

Prvé číslo v zátvorkách udáva pozíciu v riadku a druhé v stĺpci. Napríklad pixel *p(4,2)* bude uložený na miesto pixelu *p(2,4)*, ako je zobrazené na Obrázok 8.



Obrázok 8. Znázornenie pôvodného obrázku a výsledného obrázku so zamenenými osami. Červené miesto označuje pixel a jeho polohu v pôvodnom a upravenom obrázku.

4.3.2 Sekvenčná implementácia

Je skutočne triviálna a predstavuje iba iterovaný priechod poľom a preukladanie do ďalšieho poľa pri zámene súradníc.

4.3.3 Implementácia v CUDE

Pri tomto algoritme je vhodná dvojrozmerná konfigurácia vlákien v bloku, pretože uľahčuje indexáciu pri výbere a ukladaní hodnôt z pamäti.

```
1. __global__ void Kernel(uchar* d_in, uchar* d_out, int sizex, int sizey )
2. {
3.     int idx = blockDim.x * blockIdx.x + threadIdx.x;
4.     int idy = blockDim.y * blockIdx.y + threadIdx.y;
5.     d_out[idx * sizey + idy] = d_in[idy * sizex + idx];
6. }
```

Takto vyzerá kernel pre CUDU. Premenné `idx` a `idy` predstavujú indexy pixelu vo výslednom obrázku. Pri implementácii je potrebné si uvedomiť, že výsledný obrázok už nemá rozmery pôvodného obrázku, pretože sa vymenili osi obrázku.

4.3.4 Výsledky

	512 * 512	1024 * 1024	2048 * 2048
CPU	1,01	6,59	47,81
GPU	0,0832	0,2024	0,7522
Rel. Zrýchlenie	12,1	32,6	63,6

Tabuľka 3.

CUDA je architektúra, ktorá zakladá svoju silu predovšetkým na veľkom počte výpočetných jednotiek a preto je jej využitie vhodné v aplikáciách, ktoré vyžadujú veľa rovnakých

numerických výpočtov. Z výsledkov je vidno, že paralelizovaný algoritmus je relatívne neefektívny, pretože zariadenie takmer nevyužíva výpočetné jednotky zariadenia, ale pracuje iba s pamäťou. Preto nie je tento algoritmus veľmi efektívny z hľadiska časovej náročnosti v pomere ku výpočtovej náročnosti oproti iným, ktoré sú v tejto práci spracované. Ďalším faktorom, ktorý prispieva k menej efektívnej paralelizácii algoritmu je, že pri preukladaní pixelov v CUDE nepristupuje pri načítavaní k pixelom postupne podľa poradia. Tento fakt znižuje efektivitu prístupu do globálnej pamäte, zápis do pamäte je už usporiadaný správne.

4.4 Detekcia hrán v obraze

4.4.1 Teória

Táto metóda zvýrazňuje všetky hrany, ktoré sa v obrázku nachádzajú. Hrana je miesto prudkej zmeny jasu pixelov. Operátor počíta zmenu intenzity jasu pre dané miesto v obrázku a výsledok určuje, ako rýchlo alebo ako pomaly sa v danom mieste jas mení: pri pomalej zmene je výsledkom čierna farba, pri rýchlej zmene je výsledkom farba biela a táto označuje, že sa v danom mieste s veľkou pravdepodobnosťou nachádza hrana. Pre spracovanie bol zvolený Sobelov operátor [4] počíta zmenu jasu v dvoch smeroch – horizontálnom a vertikálnom. Je definovaný dvoma maticami o veľkosti 3*3, jedna je určená pre počítanie zmeny jasu pre horizontálny smer. Matice sú označené M_x pre horizontálny smer a M_y pre vertikálny smer.

$$M_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \tag{4}$$

$$M_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & -2 & +1 \end{bmatrix} \tag{5}$$

Hodnota zmeny jasu pre daný smer, ktorú označíme ako G_x a G_y (označenie z anglického gradient[7]) je vypočítaná ako súčet súčinov prvkov matice Sobelovho filtru a matice pixelov 3x3 z pôvodného obrázku, ktoré označíme ako A.

$$G_x = M_x * A \tag{6}$$

$$G_y = M_y * A \tag{7}$$

Výslednú hodnotu farby pixelu, ktorá je označená ako G, vypočítame ako druhú odmocninu súčtu druhých mocnín G_x a G_y.

$$G = \sqrt[2]{G_x^2 + G_y^2} \tag{8}$$

[7] Označenie smerovej zmeny intenzity alebo farby v obrázku.

4.4.2 Sekvenčná implementácia

Pri transformácii je potrebné iterovane prejsť pole pixelov s výnimkou pixelov v prvom a poslednom riadku a v prvom a poslednom stĺpci, pretože pri výpočte by matica Sobelovho filtru zasahovala aj za hranicu obrázku. Pre zjednodušenie sú tieto pixely z výpočtov vynechané. V ďalšom kroku je potrebné pre každý pixel vykonať dva cykly, ktoré postupne prejdú všetky pixely obklopujúce daný pixel vrátane (maticu 3x3) a vynásobia ich hodnoty s príslušným číslom v matici filtru, výsledné hodnoty každej z matíc sa sčítavajú, tento proces je potrebné urobiť dva razy pre obe matice filtru. Výsledná hodnota pixelu je vypočítaná pomocou funkcie `sqrt()`, ktorej parametrom predávam súčet druhých mocnín súčtov oboch matíc a funkcia vráti ich druhú odmocninu. Ďalej je potrebné zaistiť, že sa výsledná hodnota pixelu bude nachádzať v intervale <0, 255>. Hodnoty vyššie ako 255 sa znižujú na túto hranicu a hodnoty menšie ako 0, sa zarovnajú na 0. Využívam na to funkcie pracujúce s terárnymi operátormi, `max()`, ktorá vracia z dvojice čísiel to väčšie a `min()`, ktorá vracia z dvojice čísiel to nižšie.

4.4.3 Implementácia v CUDE

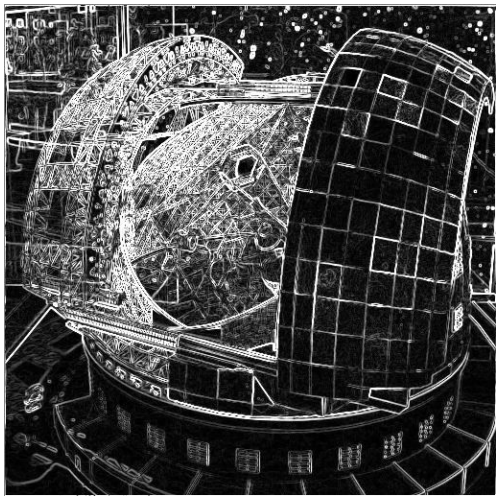
Pri implementácii pre CUDU využívam konštantnú pamäť na uloženie matíc filtru, do ktorej je pred spustením kernelu podobne ako do globálnej pamäte potrebné tieto matice nakopírovať. Konštantná pamäť je vhodná pre toto použitie pretože je prispôsobená pre broadcast dát medzi vlákna. Princíp fungovania algoritmu je veľmi podobný ako pri sekvenčnom spracovaní. Rozloženie vlákien v bloku nahrádza iterovaný prechod poľom, každé vlákno spracováva jeden pixel, tak že načíta a násobí pixely z jeho príslušnej matice so Sobelovou maticou. Kód kernelu sa nachádza v prílohe IV.

4.4.4 Výsledky

	512 x 512	1024 x 1024	2048 x 2048
CPU	9,04	36,36	145,73
GPU s globálnou pam.	0, 1062	0,2908	1,0330
GPU s konštantnou pam.	0, 0914	0,2516	0,8931
Rel. Zrýchlenie	98,9	144,5	163,2

Tabuľka 4.

V tabuľke sú zobrazené časy sekvenčného algoritmu a paralelného algoritmu bez a s využitím konštantnej pamäte. Na výsledných časoch je vidno, že algoritmus je veľmi dobre paralelizovateľný a zrýchlenie je skutočne vysoké aj napriek nevyužitiu zdieľanej pamäte. Využitie konštantnej pamäte a jej prispôsobenie na broadcast robí paralelný algoritmus ešte efektívnejším. Zrýchlenie je opäť závislé na rozlíšení a spolu s ním stúpa.



Obrázok 9. Ukážkový obrázok po aplikovaní algoritmu detekcie hrán.

4.5 Bilineárna interpolácia

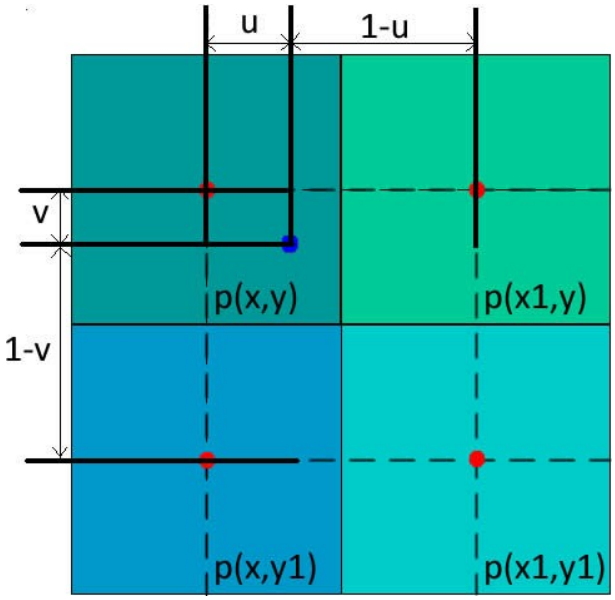
4.5.1 Teória

Algoritmus bilineárnej interpolácie[8][5] je rozšírením lineárnej interpolácie do dvoch smerov. V počítačovej grafike je využívaná ako jedna z najzákladnejších metód prevzorkovávania[9].

Bilineárna interpolácia je v počítačovej grafike používaná predovšetkým pre zistenie hodnoty pixelu v novom obrázku, na základe hodnôt pixelov v pôvodnom obrázku. Pri tomto algoritme sa využívajú 4 pixely z pôvodného obrázku, pomocou ktorých sa vypočíta hodnota pixelu v novom obrázku. Funkcia $p(x, y)$ vracia hodnotu pixelu na pozícii x a y, u je vzdialenosť hľadaného pixelu od $p(x, y)$, $1 - u$ je vzdialenosť od pixelu $p(x1, y)$, v je vzdialenosť hľadaného pixelu od $p(x, y1)$, $1 - v$ je vzdialenosť od pixelu $p(x1, y1)$. Tieto vzťahy vyjadruje Obrázok 10.

[8] Nájdenie hodnoty funkcie v danom intervale na základe iných hodnôt, ktoré v tomto intervale poznáme.

[9] Z anglického resampling, ktoré označuje metódy spojené so zmenou veľkosti alebo vlastnosti obrazu, ktoré majú za následok potrebnú zmenu jeho vzorkovania, alebo zmenu textúry, ktoré sa prispôsobujú jeho novým rozmerom.



Obrázok 10.Červené body označujú stredy pixelov. Modrý bod označuje miesto, ktorého hodnotu chceme zistiť.

Vzťah pre výpočet hodnoty hľadaného pixelu, ktorý označíme S, môžeme vyjadriť ako:

$$S \approx (1-u)*(1-v)*p(x,y) + (1-u)*(v)*p(x,y1) + (u)*(1-v)*p(x1,y) + (u)*(v)*p(x1,y1) \quad (9)$$

Vo vzťahu je vyjadrená závislosť vzdialenosti hľadaného pixelu od štyroch najbližších pixelov a koeficient, ktorým sa násobí daný pixel. Ten je tým väčší, čím je jeho vzdialenosť k stredu daného pixelu menšia.

4.5.2 Sekvenčná implementácia

V praktickej implementácii je potrebné vykonať výpočet pre každý pixel nového obrázku. Počet jednotlivých výpočtov je teda rovný počtu pixelov v obrázku, pre jednoduchšie indexovanie opäť používam dva cykly, jeden pre pohyb v riadku a druhý pre pohyb v stĺpci. Kód pre výpočet jedného pixelu vyzerá nasledovne:

```
1.  float ratio_x = (float)(size_x - 1) / (float)(size_x_res - 1);
2.  float ratio_y = (float)(size_y - 1) / (float)(size_y_res - 1);
3.  float x = (float)i * ratio_x;
4.  float y = (float)j * ratio_y;
5.  int x1 = floorf(x);
6.  int y1 = floorf(y);
7.  int x2 = min((x1 + 1), (size_x - 1));
8.  int y2 = min((y1 + 1), (size_y - 1));
9.  float u = x - (float)x1;
10. float v = y - (float)y1;
11. float value = (1-u)*(1-v)*(uchar)image(x1,y1)+(1-u)*v*(uchar)image(x1,y2)+u*(1-
    v)*(uchar)image(x2,py1) + u*v*(uchar)image(x2,y2);
12. new_image(i,j) = (uchar)value;
```

Premenné i a j sú súradnice spracovávaného pixelu. Ako prvé je treba určiť presnú pozíciu spracovávaného pixelu v pôvodnom obrázku, tým že v riadku 1 a 2 určíme pomer medzi najvyšším indexom v pôvodom a novom obrázku. V riadkoch 3 a 4 premeníme pomocou pomeru nové súradnice na súradnice v pôvodnom obrázku, vďaka ktorým v riadkoch 5 a 6 určíme prvú dvojicu súradníc, tým že indexy v pôvodnom obrázku zarovnáme na ich celočíselnú časť. Druhú dvojicu vypočítame tak, že prvú dvojicu zvýšime o jedna, vlastne sa teda posunieme o jeden stĺpec doprava a jeden riadok nižšie. Pre prípad, že prvé súradnice odkazujú na posledný riadok prípadne stĺpec, budú druhé súradnice odkazovať na miesto mimo obrázku, preto používam funkciu `min()`, ktorá výsledné súradnice v prípade prekročenia udržia na maximálnej hodnote súradníc. Ďalej je potrebné určiť hodnotu vzdialenosti súradníc pixelu od prvej dvojice zistených súradníc. Pomocou tejto vzdialenosti v riadku 11 vypočítame štyri koeficienty, ktorými vynásobíme štyri pixely obklopujúce hľadané miesto, súčet týchto súčinov je hodnota farby nového pixelu.

4.5.3 Implementácia v CUDE

4.5.3.1 Naivná implementácia

Tento algoritmus som spracoval dvoma spôsobmi, prvé spracovanie používa pre výpočet farby pixelu algoritmus popísaný v podkapitole 4.5.2. Zmenou oproti tomuto algoritmu je využitie indexovania vlákien v bloku a indexovanie blokov na nahradenie cyklov. Kód kernelu sa nachádza v prílohe V.

4.5.3.2 Implementácia s použitím textúr

V druhej implementácii tohto algoritmu využívam textúrovaciu pamäť zariadenia, ktorá je popísaná v podkapitole 3.3.3.3. Textúrovaciu pamäť je potrebné najprv nastaviť:

```
cudaChannelFormatDesc Channeldesc = CudaCreateChannelDesc(32,0,0,0,cudaChannelFormatKindFloat);
```

Textúrovacia pamäť využíva pre uloženie dát pole typu `cudaArray`, pre alokovanie tohto poľa je potrebné vytvoriť inštanciu triedy `cudaChannelFormatDesc`, pomocou ktorej sa nastavuje alokovaná pamäť, v závislosti na type dát. V konštruktore predávam parametrom hodnoty o bitovej hĺbke jednotlivých farebných zložiek pixelu a enumerátor `cudaChannelFormatKind`, ktorý bude alokovať pole pre hodnoty s desatinnou čiarkou, pretože pri využívaní interpolácie pomocou zariadenia sú dáta vo formáte float nutnosťou. Pre alokovanie je potrebné využiť funkciu `CudaMallocArray()`, ktorá prijíma parametrami ukazovateľ na pole, inštanciu `channelDesc`, šírku a výšku obrázku. Pre kopírovanie medzi hostom a zariadením využívam funkciu `cudaMemcpyToArray()`. Pre prístup k dátam v textúrovacej pamäti je ešte potrebné vytvoriť referenciu na textúru pomocou štruktúry `texture`.

```
texture<float, cudaTextureType2D, cudaReadModeElementType> tex;
```

Pri vytváraní referencie textúry predávam konštruktoru parametre, ktoré hovoria že textúra bude pracovať s jednozložkovým typom float a textúra bude dvojrozmerná. Posledný parameter je enumerátor ReadMode, ktorý môže byť nastavený na `cudaReadModeNormalizedFloat`, ktorý zariadeniu hovorí, že dáta sa budú konvertovať do normalizovanej formy, teda mapovať na float hodnoty 0 až 1, prípadne -1 až 1, `cudaReadModeElementType` nastavuje načítanie dát bez konverzie.

```
1.  tex.addressMode[0] = cudaAddressModeClamp;
2.  tex.addressMode[1] = cudaAddressModeClamp;
```

Tieto riadky slúžia pre nastavenie adresovania v textúre, prvý riadok nastavuje adresáciu pre os *x* a druhý pre os *y*, enumerátor AddressMode definuje čo sa stane ak sa súradnice dostanú mimo rozsahu textúry. `CudaAddressModeClamp` je adresovací mód, ktorý pri prekročení rozsahu textúry načíta hodnotu z poslednej prístupnej adresy z textúry, najbližšej zadaným súradniciam

```
tex.filterMode = cudaFilterModeLinear;
```

Ďalším nastavením je filterMode, ktorý textúre nastavuje typ filtrovania výstupných hodnôt. `cudaFilterModeLinear` nastavuje, že súradnice predané textúre budú spracované a výsledná hodnota bude bileneárnou interpoláciou štyroch texelov[10] najbližších k zadaným súradniciam.

```
tex.normalized = false;
```

Posledným potrebným nastavením je normalizácia súradníc, ak je hodnota nastavená na `false`, žiadna konverzia sa nevykoná. V prípade, že hodnota bude `true`, rozsah súradníc vstupných dát sa v textúre bude mapovať na float hodnoty 0 až 1.

```
cudaBindTextureToArray(tex, d_inArray, channelDesc);
```

Po vytvorení referencie na textúru a jej nastavení, je potrebné túto textúru prepojiť s poľom dát, pomocou príkazu `cudaBindTextureToArray`.

V kernely je potrebné určiť súradnice spracovávaného pixelu, vzhľadom na pôvodný obrázok, rovnakým postupom ako je popísaný v podkapitole 4.5.2. Najbližšie pixely už nie je potrebné vyhľadávať, pretože zariadenie to spraví samo:

```
float pixel = tex2D(tex, x, y);
```

Funkcia `tex2D()` pomocou referencie na textúru a súradníc vykoná bilineárnu interpoláciu 4 najbližších pixelov a vráti hodnotu nového pixelu.

[10] Vzniklo spojením slov Texture Element a predstavuje jeden prvok z poľa, ktoré tvorí textúru.

4.5.4 Výsledky

	512 x 512	1024 x 1024	2048 x 2048
CPU	41,45	167,7	676,45
GPU	0,1945	0,6624	2,4732
Rel. Zrýchlenie	213,1	253,2	273,5

Tabuľka 5.

V tabuľke sú uvedené časy namerané pri vykonávaní naivnej implementácie, aj v tomto prípade je výsledné zrýchlenie vysoké aj bez použitia optimalizácií.

	512 x 512	1024 x 1024	2048 x 2048
CPU	41,45	167,7	676,45
GPU	0,1364	0,4082	1,5078
Rel. Zrýchlenie	303,9	410,8	448,6

Tabuľka 6.

V tabuľkeTabuľka 6. sú uvedené časy namerané pri sekvenčnom natívnom spracovaní a časy pri optimalizovaní algoritmu pomocou použitia textúr a filtrovania prostredníctvom zariadenia. Relatívne zrýchlenie sa oproti natívnemu paralelnému algoritmu zvýšilo veľmi podstatne. K zvýšeniu napomáha textúrovacia cache pamäť a aj fakt, že zložité viackrokové numerické operácie z natívneho algoritmu nahradilo filtrovanie prostredníctvom zariadenia, ktoré má na tieto účely špecializované, optimalizované jednotky.

4.6 Gaussovho rozostrovanie

4.6.1 Teória

Gaussovo rozostrovanie [6] je rozmazávanie obrazu pomocou Gaussovej funkcie a v praxi sa využíva na zjemnenie detailov, alebo odstránenie šumu v obraze. Gaussova funkcia sa využíva na výpočet transformácie, ktorá sa aplikuje na každý pixel. V princípe sa transformácia pixelu zakladá na výpočte váženého priemeru jeho susedných bodov, kde váha je závislá na vzdialenosti od transformovaného pixelu. Pre transformáciu v dvojrozmernom priestore sa Gaussova funkcia rovná:

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{10}$$

Kde x je vzdialenosť od transformovaného pixelu v horizontálnej ose, y je vzdialenosť vo vertikálnej ose, a σ je smerodajná. Pri dosadení ktoréhokoľvek pixelu je táto rovnica nenulová, inými slovami, každý pixel v obraze sa môže podieľať na výpočte váženého priemeru, avšak pixely vo vzdialenosti 3σ dosadené do Gaussovej funkcie nadobúdajú funkčnú hodnotu natoľko blízku hodnote 0, že môžu byť považované za 0 a ich zohľadnenie na výpočte je neefektívne. Z toho vyplýva, že matica s normalizovanými hodnotami Gaussovej funkcie vzhľadom na stred

by mala mať rozmery $\lceil 6\sigma + 1 \rceil$ x $\lceil 6\sigma + 1 \rceil$, kde $\lceil 6\sigma + 1 \rceil$ vracia najnižšiu celočíselnú hodnotu vyššiu, alebo rovnú $6\sigma + 1$.

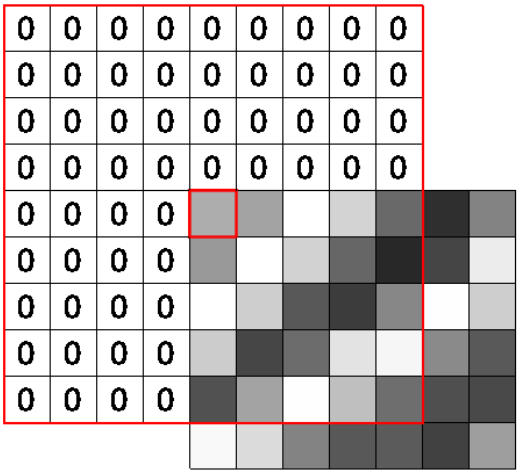
Pre transformáciu pixelu je potrebná matica pomocou, ktorej je možné vypočítať vážený priemer pixelov v danej časti obrázku. Pre vytvorenie matice som pripravil algoritmus, ktorý naplní dvojrozmerné pole normalizovanými hodnotami Gaussovej funkcie:

```
1.  const int size = 9;
2.  float kernel[size][size];
3.  int r = size / 2;
4.  float s = ((float)size - 1.f) / 6.f;
5.  for (int y = -r; y <= r; y++) for (int x = -r; x <= r; x++)
6.      kernel[r + x][r + y] = (1 / (2*pi*s*s)) * exp(-(x*x + y*y) / (2*s*s));
```

Transformovaný prvok by mal mať vo funkcii najväčšiu váhu, preto musí byť rozmer matice nepárny. V riadku 3 si vypočítame polomer matice, inými slovami to je počet pixelov pred transformovaným pixelom, alebo za ním. Hodnotu sigma si určíme na riadku 4, pričom výpočet zaručuje splnenie podmienky: 6σ + 1 je menšia alebo rovná rozmeru matice. V riadku 6 je spracovaná konečná Gaussova funkcia.

4.6.2 Sekvenčná Implementácia

Pri implementácii je potrebné sa rozhodnúť, či bude matica zasahovať aj mimo obrázku a budú teda spracované všetky pixely tak ako to ilustruje Obrázok 11, alebo budú pri okrajoch obrazu rady pixelov nespracované a implementácia bude jednoduchšia a pri CUDE aj efektívnejšia. Avšak polomer matice je pomerne veľký a nespracované pixely by tvorili nezanedbateľnú časť obrázku, preto som v implementácii zohľadnil aj pixely pri okrajoch.



Obrázok 11. Obrázok ilustruje maticu bodov, ktorá zasahuje mimo oblasti obrázku, pri spracovaní prvého pixelu.

Kód pre procesor vyzerá nasledovne:

```
1.  for(int y=0;y<sizey;y++){
2.      for(int x=0;x<sizex;x++){
3.          value = 0; pixel = 0;
4.          for(int j = -g_radius; j<= g_radius; j++){
5.              for(int i = -g_radius; i<= g_radius; i++){
6.                  if(x+i < 0 || y+j < 0 || x+i > sizex-1 || y+j > sizey-1)
7.                      value = 0;
8.                  else
9.                      value = (uchar) image(x+i, y+j);
10.                 pixel += value * gauss_kernel[g_radius+i][g_radius+j]; } }
11.                 new_image(x, y) = pixel;
12.     }
13. }
```

Kód bude spustený pre každý pixel v obrázku. Posun v stĺpci, teda po osi *y* zabezpečuje prvý cyklus a posun v riadku teda v osi *x* zabezpečuje druhý cyklus. V riadkoch 4 a 5 sú ďalšie dva cykly, ktoré majú za úlohu posun od transformovaného pixelu o polomer matice. Pretože súradnice, s ktorými pracujeme patria stredu matice, je potrebné aby cykly začínali na zápornej hodnote polomeru a pokračovali až do kladnej hodnoty polomeru a opäť prvý slúži na posun v matici vertikálne, a druhý horizontálne. V riadku 6 je potrebné otestovať, či matica transformovaného pixelu nezasahuje mimo okraju obrázku. Ak áno, tak hodnota pixelu, ktorý je mimo obrázku bude rovná 0, v prípade, že pixel je v obrázku, tak je jeho hodnota načítaná. V riadku 10 sa vynásobí hodnota pixelu s príslušnou normalizovanou hodnotou v Gaussovej matici a tento medzivýsledok sa pripočíta k súčtu týchto operácií ostatných pixelov danej matice. Po vykonaní tejto operácie pre celú maticu, sa súčet operácií rovná hodnote transformovaného pixelu.

4.6.3 Implementácia v CUDE

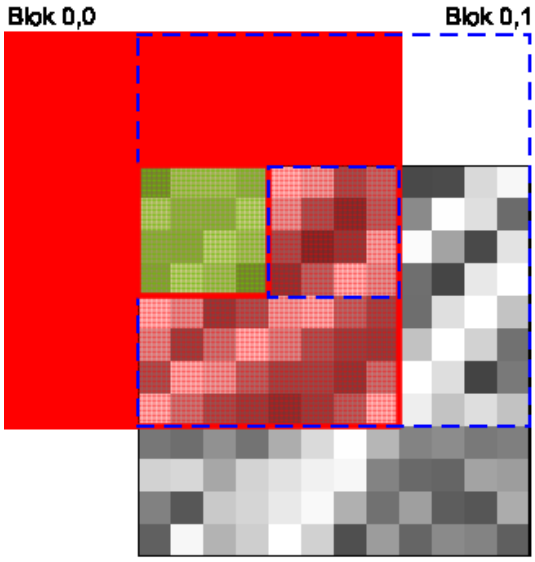
V CUDE som naimplementoval tri riešenia tohto algoritmu, postupne si ich popíšeme od najjednoduchšieho. Každý z algoritmov využíva dvojrozmernú konfiguráciu vlákien v bloku a aj blokov v mriežke, kvôli jednoduchšiemu indexovaniu pixelov. Rozmery bloku výsledných riešení sú 8x8 vlákien, kvôli efektívnemu využitiu zdieľanej pamäte, ktorá bude popísaná nižšie.

4.6.3.1 Naivná implementácia

Ide o najjednoduchšie spracovanie algoritmu pre procesor do prostredia CUDY bez optimalizácií. Algoritmus je založený na rovnakom princípe ako sekvenčný, popísaný v podkapitole 4.6.2. Kód kernelu sa nachádza v prílohe VII. Keďže algoritmus musí ošetriť aj prípady keď Gaussova matica zasahuje mimo poľa dát, budú v algoritme podmienky, ktoré vetvia program. Podmienky, ako bolo spomínané v podkapitole 3.3, delia program na vetvy a tie spôsobujú divergenciu vlákien, ktorá dokáže brzdiť výkon zariadenia. Podmienky spojené spojkami „alebo“ z pôvodného algoritmu som rozdelil na jednotlivé samostatné a čas vykonávania sa znížil takmer na polovicu.

4.6.3.2 Implementácia s využitím zdieľanej pamäte

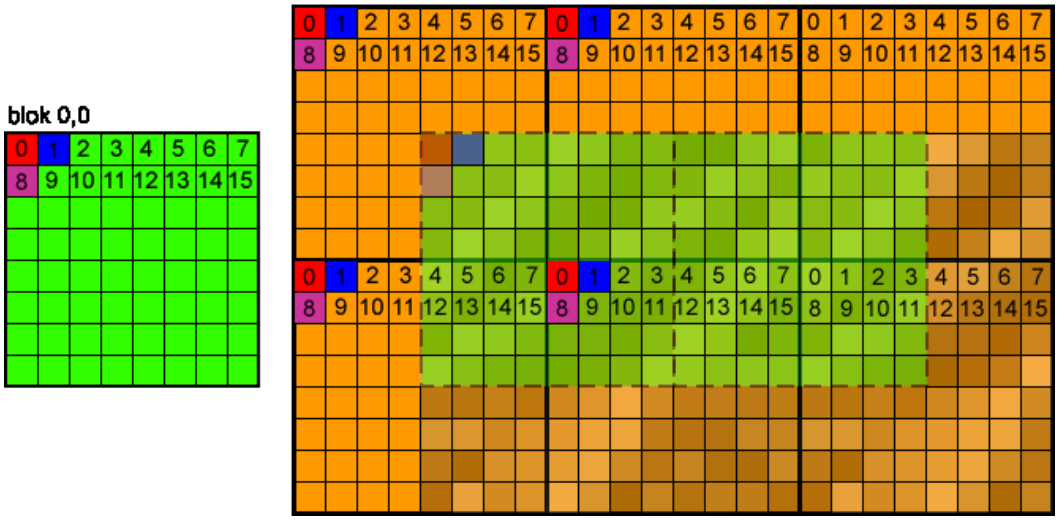
Prvotný algoritmus s využitím zdieľanej pamäte bol založený na princípe, že každé vlákno načíta jeden pixel do zdieľanej pamäte, ale transformovanie robia iba vlákna vzdialené od okraju bloku o polomer Gaussovej matice. Tieto bloky boli ku sebe pritlačené stredom, čiže aktívnou časťou, tak ako to ilustruje Obrázok 12.



Obrázok 12.Ilustrácia princípu pôvodného algoritmu s využitím zdieľanej pamäte. Väčší červený štvorec označuje blok vlákien a pixely, ku ktorým vlákna pristupujú. Červené šrafovanie označuje vlákna, ktoré sú pri transformácii neaktívne. Zelené šrafovanie označuje aktívnu časť bloku, inými slovami vlákna, ktoré budú im prislúchajúce pixely transformovať. Modrý prerušovaný štvorec ilustruje rozloženie blokov.

Tento algoritmus sa ale ukázal ako veľmi neefektívny, práve z dôvodu, že veľká časť bloku vlákien je pri výpočtoch neaktívna a pri zväčšení aktívnej časti bude rásť aj počet neaktívnych vlákien. Z týchto dôvodov bol tento koncept zavrhnutý.

Druhotný algoritmus využíva plný počet vlákien. Jedno vlákno načíta až štyri pixely [7], čím sa mi podarilo úplne využiť vlákna v bloku. Princíp načítania pixelov je ilustrovaný na Obrázok 13.



Obrázok 13. Blok vlákien (zelený štvorec naľavo), v ktorom sú zobrazené vlákna 0 až 15.Obdĺžnik napravo predstavuje ľavý horný roh obrázku a dva bloky vlákien. Farebné štvorčeky 0,1 a 8 zobrazujú pixely ktoré bude načítať 0,1 a 8 vlákno z bloku 0,0.

Pri konfigurácii vlákien v bloku je potrebné, aby bol blok štvorcový a podľa polomeru Gaussovej matice je potrebné nastaviť šírku na dva polomery. Pre zachovanie vysokej efektivity warpov, je potrebné aby počet vlákien v bloku bol celočíselným násobkom počtu vlákien vo warpe, aby sme sa vyhli neaktívnym vláknam. V implementácii používam maticu o veľkosti 9x9, polomer je 4, šírka bloku je následne 8x8, podobne ako na Obrázok 13.

Kód kernelu sa nachádza v prílohe VIII. Na začiatku je potrebné vyalokovať zdieľanú pamäť o veľkosti bloku a časti pixelov okolo bloku o šírke polomeru Gaussovej matice. Zdieľaná pamäť je alokovaná ako jednorozmerné pole, na úkor jednoduchšieho indexovania to minimalizuje bankové konflikty a mnohonásobne zvyšuje rýchlosť prístupu k pamäti. Následne je potrebné zdieľanú pamäť naplniť tak, že každé vlákno načíta po sebe štyri pixely podľa svojich súradníc. Vlákno na pozícii *x, y* načíta pixely do zdieľanej pamäte na pozíciách *(x - polomer, y - polomer), (x + polomer, y - polomer), (x – polomer, y + polomer) a (x + polomer, y + polomer)*. V prípade, že súradnice odkazujú za hranicu obrázku, podmienky zaručia že na príslušné miesto v zdieľanej pamäti vlákno načíta hodnotu 0. Po načítaní všetkých 4 pixelov je potrebné synchronizovať vlákna, aby sme sa uistili že v čase transformovania pixelov, už všetky vlákna načítali svoje pixely a nenastane prípad, že vlákno bude pracovať s hodnotami, ktoré ešte neboli načítané. V ďalšom kroku je potrebné v dvoch cykloch prejsť maticu pixelov, obklopujúce transformovaný pixel a vynásobiť v každej iterácii pixel z matice s hodnotou z Gaussovej matice na príslušnej pozícii. Výsledky je potrebné sčítavať a hodnota po skončení oboch cyklov je hodnota transformovaného pixelu.

4.6.3.3 Implementácia s využitím zdieľanej pamäte a textúry

Algoritmus v podkapitole 4.6.3.2, je skutočne rýchly ako bude možné vidieť vo výsledkoch, avšak využíva podmienky, ktoré vytvárajú divergentné vlákna. V tejto kapitole si popíšeme algoritmus, ktorý už podmienky nevyužíva.

Ako bolo spomenuté v podkapitole 3.3.3.3, textúrovacia pamäť poskytuje rôzne módy načítavania z pamäte zariadenia. Jednou z výhod načítavania pomocou textúr je možnosť nastaviť textúrovaciu pamäť tak, že pri prístupe k pixelu, ktorý v textúre neleží, textúra vráti poslednú hodnotu v textúre na mieste najbližšom zadaným súradniciam. Vďaka tejto schopnosti je možné ošetriť prístupy mimo poľa obrázku. V podkapitole 4.5.3 je popísané nastavenie textúrovacej pamäte a preto ho nebudem uvádzať znovu celé ale ukážem iba nastavenia, ktoré sú odlišné.

```
desc = cudaCreateChannelDesc(8,0,0,0,cudaChannelFormatKindUnsigned);
```

Pre textúru je potrebné vytvoriť inštanciu `cudaCreateChannelDesc` a pretože budem pracovať s typom unsigned char, je potrebné konštruktoru predať bitovú hĺbku a enumerátor, ktorý nastavuje, že typ bude bezznamienkový.

```
tex.filterMode = cudaFilterModePoint;
```

Pri tomto algoritme nie je potrebné žiadne filtrovanie a preto filtrovací mód nastavíme na `cudaFilterModePoint`, ktorý zaručí, že textúra vráti texel najbližší k zadaným súradniciam.

Kód algoritmu je až na časť s načítavaním do zdieľanej pamäte rovnaký ako algoritmus z podkapitoly 4.6.3.2 a preto si popíšeme iba načítanie. Celý kernel sa nachádza v prílohe IX.

```
1.  s_data[ty*g_smem_x+tx]= tex2D(tex, idx - g_radius, idy - g_radius);
2.  s_data[ty*g_smem_x+tx+bdx]= tex2D(tex, idx + g_radius, idy - g_radius);
3.  s_data[(ty+bdy)*g_smem_x+tx] = tex2D(tex, idx - g_radius, idy + g_radius);
4.  s_data[(ty+bdy)*g_smem_x+tx+bdx] = tex2D(tex, idx + g_radius, idy + g_radius);
```

V každom riadku načítam jeden pixel z textúry pomocou funkcie `tex2D()` a ukladám ho na príslušné miesto do zdieľanej pamäte rovnako ako v algoritme bez použitia textúr.

4.6.4 Výsledky

Výsledky sú uvádzané pre algoritmy pracujúce s Gaussovou maticou s rozmermi 9x9.

	512 * 512	1024 * 1024	2048 * 2048
CPU	111,15	444,79	1780,97
GPU	1,3731	5,3917	22,8165
Rel. Zrýchlenie	80,9	82,5	78,1

Tabuľka 7.

V tabuľke sú uvedené časy naivnej implementácie tohto algoritmu. Doba vykonávania ukazuje, že tento algoritmus je skutočne náročný a v sekvenčnom algoritme sú časy skutočne vysoké. Aj časy na zariadení sú pomerne vysoké oproti iným algoritmom, ktoré boli spracované a rozdiely medzi časmi v rôznych rozlíšeniach sú takmer rovnaké, pretože čas spustenia a inicializácie kernelu je zanedbateľný vzhľadom na čas strávený nad výpočtami.

	512 x 512	1024 x 1024	2048 x 2048
CPU	111,15	444,79	1780,97
GPU	0,3499	1,2994	5,024
Rel. Zrýchlenie	317,7	342,3	354,5

Tabuľka 8.

V tabuľke sú uvedené časy vykonávania algoritmu s využitím zdieľanej pamäte a naivného sekvenčného algoritmu. Využitie zdieľanej pamäte skutočne prináša vysoké zdvihnutie rýchlosti, výsledné zrýchlenie sa oproti naivnej paralelnej implementácii sa zvýšilo viac ako 4krát.

	512 x 512	1024 x 1024	2048 x 2048
CPU	111,15	444,79	1780,97
GPU	0,3108	1,1347	4,3677
Rel. Zrýchlenie	357,6	392,0	407,8

Tabuľka 9.

V tabuľkeTabuľka 9. sú uvedené časy algoritmu s použitím zdieľanej pamäte a textúr a opäť časy naivnej sekvenčnej implementácie. Pomocou zdieľanej pamäte algoritmus nepracuje s podmienkami a nevytvára divergentné vlákna. Tento fakt znižuje časy vykonávania paralelného algoritmu a zvyšuje tak relatívne zrýchlenie až na hodnoty vyššie ako 400.

5 Záver

CUDA má skutočne obrovský potenciál v rôznych sférach využitia. V práci som pracoval s metódami spracovania obrazu a ich paralelizácii. Tento druh algoritmov je dobre paralelizovateľný, pretože pre veľkú množinu dát vykonáva v podstate rovnaký úkon, alebo množinu úkonov. V správnom nasadení a využití dostupných prostriedkov dokáže paralelný kód na CUDE priniesť obrovské zrýchlenia v rádoch stoviek, tak ako to preukázali výsledky namerané na zvolených implementovaných algoritmoch. Z výsledkov je zrejmé, že efektivita spracovania na CUDE sa zvyšuje s objemom dát, ktoré sa spracovávajú, samozrejme za predpokladu, že zvolený postup je správny.

Práca na spracovaní zadaných metód bola skutočne zaujímavá a poučná a dúfam, že som technológiu CUDA a vybrané metódy zrozumiteľne popísal. Pri práci som skutočne pochopil princíp paralelizmu a v budúcnosti by som ho chcel určite využiť aj na miestach, na ktorých som jeho význam vôbec nevidel. Verím, že mnou navrhnuté riešenie paralelných algoritmov je dostatočne efektívne a správnym spôsobom využíva dostupné prostriedky, ktoré CUDA ponúka. Nasadenie môjho riešenia by som si vedel predstaviť na miestach, kde aj milisekundy hrajú podstatnú úlohu, napr. vo virtuálnych fotoalbumoch, kde by dokázali moje algoritmy spracovať aj 10 000 obrázkov za sekundu aj na bežných, dnes dostupných zariadeniach od spoločnosti NVIDIA.

Budúci vývoj mnou navrhnutých paralelných algoritmov vidím v spolupráci viacjadrového procesoru a grafického akcelerátoru, kde by procesor spracovával ťažko paralelizovateľné časti algoritmu a zariadenie by vykonávalo časti kódu, ktoré sa dajú efektívne paralelizovať.

6 Použitá literatúra

[1] CUDA. In: Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 29.5.2007, 31.3.2012 [cit. 2012-04-16]. Dostupné z: http://en.wikipedia.org/wiki/CUDA

[2] NVIDIA. *NVIDIA CUDA C Programming Guide 4.1*[online]. 18.11.2011 [cit. 2012-04-16]. Dostupné z: http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf

[3] FARBER, Rob. CUDA, Supercomputing for the Masses. *Dr.Dobbs* [online]. 15.4.2008[cit. 2012-04-16]. Dostupné z: http://www.drdobbs.com/cpp/207200659

[4] MATHEWS, James. An Introduction to Edge Detection: The Sobel Edge Detector. In: Generation5 [online]. 27.1.2002 [cit. 2012-04-23]. Dostupné z: http://www.generation5.org/content/2002/im01.asp

[5] Coding Bilinear Interpolation. In: *The Supercomputing Blog* [online]. 30.12.2011 [cit. 2012-04-16]. Dostupné z: http://supercomputingblog.com/graphics/coding-bilinear-interpolation/

[6] Gaussian blur. In: *Wikipedia*: *the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 5.5.2005, 27.3.2012 [cit. 2012-04-16]. Dostupné z: http://en.wikipedia.org/wiki/Gaussian_blur

[7] PODLOZHNYUK, Victor. NVIDIA. Image Convolution with CUDA [online]. 1.0. 6.2007 [cit. 2012-04-16]. Dostupné z: http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_64_website/projects/convolutionSeparable/doc/convolutionSeparable.pdf

7 Zoznam príloh

I. Kernel Invertovania farby

II. Kernel Zmeny jasu

III. Kernel preklápania

IV. Kernel detekcie hrán

V. Kernel naivnej bilineárnej interpolácie

VI. Kernel bilineárnej interpolácie pomocou textúr

VII. Kernel naivného Gaussovho rozostrovania

VIII. Kernel Gaussovho rozostrovania so zdieľanou pamäťou

IX. Kernel Gaussovho rozmazávania so zdieľanou pamäťou a textúrami

X. Príloha na CD

Teoretická časť:

- Práca.pdf

Praktická časť:

- Generátor Gaussovej matice
- Program s metódami spracovania obrazu

I. Kernel Invertovania farby

```
__global__ void CUDAinvertKernel(uchar* d_in, uchar* d_out, int sizex, int sizey)
{
	int idx = blockDim.x * blockIdx.x + threadIdx.x;
	int pixel = d_in[idx];
	d_out[idx] = 255 - pixel;
}
```

II. Kernel Zmeny jasu

```
__global__ void CUDAbrightenKernel(uchar* d_in, uchar* d_out, int sizex, int
sizey, int percent )
{
	int idx = blockDim.x * blockIdx.x + threadIdx.x;
	float pixel = d_in[idx];
	float bright = ((255 - pixel) / 100 * percent);
	pixel = bright + pixel;
	d_out[idx] = pixel;
}
```

III. Kernel preklápania

```
__global__ void CUDArotationKernel(uchar* d_in, uchar* d_out, int sizex, int
sizey )
{
	int idx = blockDim.x * blockIdx.x + threadIdx.x;
	int idy = blockDim.y * blockIdx.y + threadIdx.y;
	d_out[idx * sizey + idy] = d_in[idy * sizex + idx];
}
```

IV. Kernel detekcie hrán

```
__global__ void CUDAsobelKernel(uchar* d_in, uchar* d_out, int size_x, int
size_y )
{
       int idx = blockDim.x * blockIdx.x + threadIdx.x;
       int idy = blockDim.y * blockIdx.y + threadIdx.y;
       int index = idy * size_x + idx;

       int x=0, y=0;
       for( int j = -1; j <= 1; j++ ){
              for( int i = -1; i <= 1; i++ ){
                     x += d_in[j * size_x + index + i] * d_kx[1+i][1+j];
                     y += d_in[j * size_x + index + i] * d_ky[1+i][1+j];
              }
       }
    float pixel = sqrt( float(x*x) + float(y*y) );
    pixel = max(0,pixel);
       pixel = min(255,pixel);
       __syncthreads();
       d_out[index] = pixel;
}
```

V. Kernel naivnej bilineárnej interpolácie

```
__global__ void CUDAbilinearInterpolationKernel(uchar* d_in, uchar* d_out, int
size_x, int size_y, int size_x_res, int size_y_res)
{
       int idx = blockIdx.x * blockDim.x + threadIdx.x;
       int idy = blockIdx.y * blockDim.y + threadIdx.y;
       if (idx >= size_x_res || idy >= size_y_res ) return ;
       float ratio_x = (float)(size_x - 1) / (float)(size_x_res - 1);
       float ratio_y = (float)(size_y - 1) / (float)(size_y_res - 1);
       float x = (float)idx * ratio_x;
       float y = (float)idy * ratio_y;
       int x1 = floorf(x);
    int y1 = floorf(y);
       int x2 = min((x1 + 1), (size_x - 1));
    int y2 = min((y1 + 1), (size_y - 1));
       float u = x - (float)x1;
    float v = y - (float)y1;
       float value = (1-u)*(1-v)*d_in[y1 * size_x + x1] + (1-u)*v*d_in[y2 *
size_x + x1] +
              u*(1-v)*d_in[y1 * size_x + x2] + u*v*d_in[y2 * size_x + x2];
       d_out[idy * size_x_res + idx] = (int)value;
}
```

VI. Kernel bilineárnej interpolácie pomocou textúr

```
__global__ void CUDAbilinearTexture(float* d_out, int size_x, int size_y, int
size_x_res, int size_y_res)
{
	int idx = (blockIdx.x * blockDim.x) + threadIdx.x;
	int idy = (blockIdx.y * blockDim.y) + threadIdx.y;
	int index = (idy * size_x_res) + idx;
	float ratio_x = (float)(size_x - 1) / (float)(size_x_res - 1);
	float ratio_y = (float)(size_y - 1) / (float)(size_y_res - 1);
	float x = (float)idx * ratio_x;
	float y = (float)idy * ratio_y;
	if ((idx < size_x_res) && (idy < size_y_res)){
		float pixel = tex2D(tex, x, y);
		d_out[index] = pixel;
    }
}
```

VII. Kernel naivného Gaussovho rozostrovania

```
__global__ void kernel_1(uchar *d_in, uchar *d_out, int size_x, int size_y)
{
	short tx = threadIdx.x;
	short ty = threadIdx.y;
	short bx = blockIdx.x;
	short by = blockIdx.y;
	short bdx = blockDim.x;
	short bdy = blockDim.y;
	int idx = (bx * bdx) + tx;
	int idy = (by * bdy) + ty;
	int index = idy * size_x + idx;
	float value = 0;
	float pixel = 0;
	for (int j = -g_radius; j <= g_radius; j++){
		for (int i = -g_radius; i <= g_radius; i++){
			if (idx + i < 0)

				value = 0;
			else if (idy + j < 0)

				value = 0;
			else if (bx == gridDim.x - 1 && tx + i > bdx - 1)
				value = 0;
			else if (by == gridDim.y - 1 && ty + j > bdy - 1)
				value = 0;
			else value = d_in[j * size_x + index + i];
			pixel += value * d_kernel[g_radius + i][g_radius + j];
		}
	}
	d_out[index] = pixel;
}
```

VIII. Kernel Gaussovho rozostrovania so zdieľanou pamäťou

```
__global__ void kernel_2(uchar *d_in, uchar *d_out, int size_x, int size_y)
{
	__shared__ float s_data[g_smem_x * g_smem_y];
	short tx = threadIdx.x;
	short ty = threadIdx.y;
	short bx = blockIdx.x;
	short by = blockIdx.y;
	short bdx = blockDim.x;
	short bdy = blockDim.y;
	int idx = (bx * bdx) + tx;
	int idy = (by * bdy) + ty;
	int index = idy * size_x + idx;
	if ( idx - g_radius < 0 || idy - g_radius < 0 )
		s_data[ g_smem_x * ty + tx ] = 0;
	else
		s_data[ g_smem_x * ty + tx ] = (float)d_in[index - g_radius -
(size_x * g_radius)];
	if ( idx + g_radius > size_x-1 || idy - g_radius < 0 )
		s_data[ g_smem_x * ty + bdx + tx ] = 0;
	else
		s_data[ g_smem_x * ty + bdx + tx ] = (float)d_in[index + g_radius
- (size_x * g_radius)];
	if (idx - g_radius < 0 || idy + g_radius > size_y - 1)
		s_data[(ty + bdy) * g_smem_x + tx ] = 0;
	else
		s_data[(ty + bdy) * g_smem_x + tx ] = (float)d_in[index -
g_radius + (size_x * g_radius)];
	if ( idx + g_radius > size_x-1 || idy + g_radius > size_y-1)
		s_data[ (ty + bdy) * g_smem_x + bdx + tx ] = 0;
	else
		s_data[ (ty + bdy) * g_smem_x + bdx + tx ] = (float)d_in[index +
g_radius + (size_x * g_radius)];
	__syncthreads();
	float pixel = 0;
	for (int j = -g_radius; j <= g_radius; j++)
		for (int i = -g_radius; i <= g_radius; i++)
			pixel += s_data[(g_radius + ty + j) * g_smem_x + g_radius +
tx + i ] * d_kernel[g_radius + i][g_radius + j];
	d_out[index] = (int)pixel;
}
```

IX. Kernel Gaussovho rozmazávania so zdieľanou pamäťou a textúrami

```
_global__ void kernel_2D(uchar *d_out, int size_x, int size_y)
{
	__shared__ float s_data[g_smem_x * g_smem_y];
	short tx = threadIdx.x;
	short ty = threadIdx.y;
	short bx = blockIdx.x;
	short by = blockIdx.y;
	short bdx = blockDim.x;
	short bdy = blockDim.y;
	int idx = (bx * bdx) + tx;
	int idy = (by * bdy) + ty;
	int index = idy * size_x + idx;
	s_data[ty * g_smem_x + tx] = tex2D(tex, idx - g_radius, idy - g_radius);
	s_data[ty * g_smem_x + tx + bdx] = tex2D(tex, idx + g_radius, idy -
g_radius);
	s_data[(ty + bdy) * g_smem_x + tx] = tex2D(tex, idx - g_radius, idy +
g_radius);
	s_data[(ty + bdy) * g_smem_x + tx + bdx] = tex2D(tex, idx + g_radius,
idy + g_radius);
	__syncthreads();
	float pixel = 0;
	for (int j = -g_radius; j <= g_radius; j++)
		for (int i = -g_radius; i <= g_radius; i++)
			pixel += s_data[(g_radius + ty + j) * g_smem_x + g_radius +
tx + i ] * d_kernel[g_radius + i][g_radius + j];
	d_out[index] = (int)pixel;
}
```